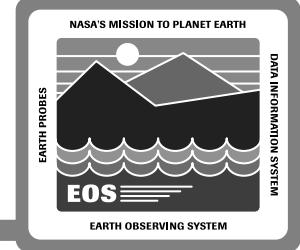


Error/Exception Handling

Nitin Vazarkar

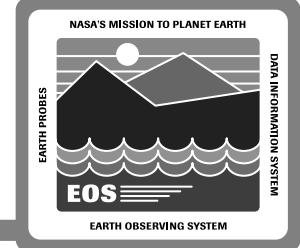
Developers Workshop
30 May 1995

Agenda



- Overview of C++ error/exception handling
 - Basics (what/how/when)
 - Benefits (why should I use it ?)
 - Rules
 - Examples
- Products that use exceptions
 - HP OODCE
 - Roguewave
- Error/Exception handling for ECS
- Summary

Introduction

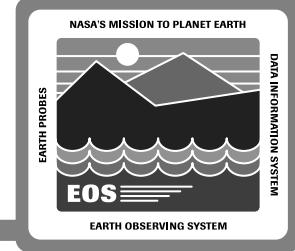


- Exceptions are run-time program anomalies (user, logic or system errors) that prevent it from continuing.
- Exception handling provides a language supported (standard) way for responding to exceptions (C++ ANSI draft X3J16).
- Exceptions are used for run-time error checking and handling !
 - throwing exceptions and returning errors are mutually exclusive

Benefits:

- Reduces size and complexity of the program
 - Focus attention on normal behaviour
- Consistent error handling for a product/project
- Exception hierarchy may be created for scoping the error handling.
- Fine tuning is possible for a specific task or project.

Language support



C++ provides a set of keywords to support exception handling

try

- **caller encloses a set of statements which may raise an exception in this block (try-block).**
- **if no exception is raised, entire try-block is executed.**

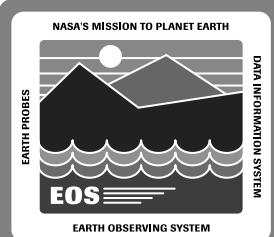
catch

- **used to trap exceptions raised in the try block (catch-handler)**
- **placed after the try-block**
- **multiple catch handlers are allowed to catch different exceptions**

throw

- **exception is raised using this statement**
- **a class or type is ‘thrown’ which is used to match with the catch-handler**

Example (try/catch)



```
// declaration of the exception class
class MathErr {...};

class Overflow : public MathErr {
public: Overflow(char Operation, double arg1, double arg2)
};

void foo(float x, float y) // some routine. maybe called from main
{
    try { // start of try block. may have several statements
        float z = Multiply (x, y); // call function in Math library
        result = Divide(z, 3.45); // call function in math library
        ... // other functions in the math library
    } // end of try block

    // install catch handlers here. start of catch block
    catch (Overflow of) { // catch exception of type Overflow
        cerr << "caught overflow exception " << (char *)oo << endl;
    }
    catch (...) { // catch all exceptions
        cerr << "unknown exception" << endl;
    }

    // end of catch blocks. if no exceptions are thrown or after
    // handling a thrown exception control comes here
    cout << "after try block" << endl;
}
```

{

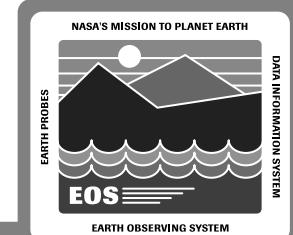
try
block

{}

catch
handlers
(block)

{}

post
catch
block



Examples (throw)

```
// throwing exception
```

```
float Multiply(float arg1, float arg2)
{
    ...
    if (overflow condition)
        throw Overflow('x' /* multiply operation */,
                      arg1, arg2);
    // if exception is thrown control does not come here
    ...
}
```

```
// Passing exception to the next
```

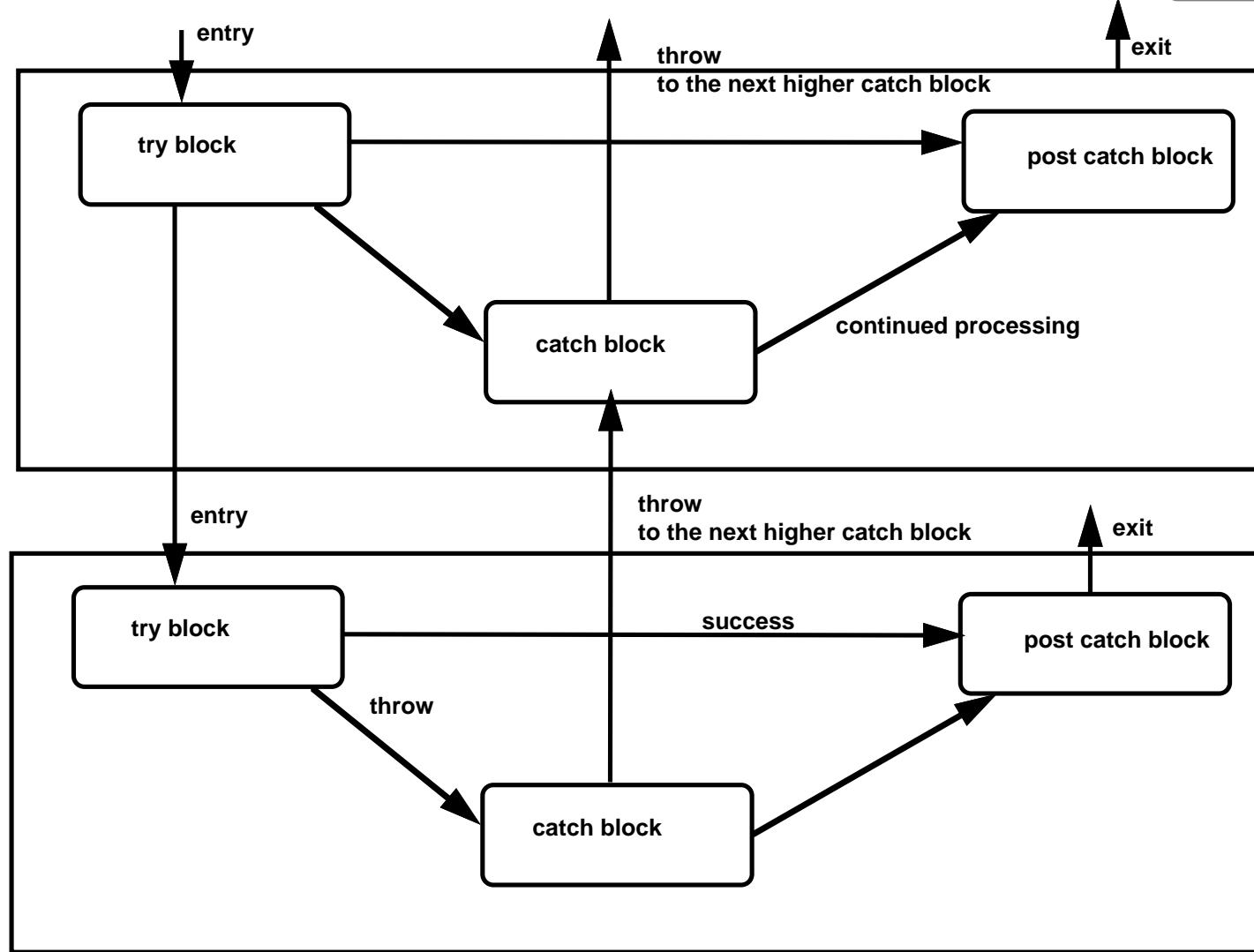
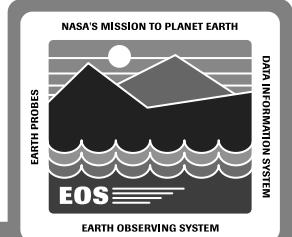
```
// higher catch handler
catch (...)

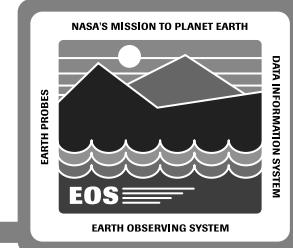
{
    // respond partially
    throw; // rethrow for next level
}
```

```
// Other examples
```

```
throw "Panic: I'm not well !!"; // caught by 'catch (const char* message)' handler
throw Underflow('/', arg1, arg2); // caught by 'catch (Underflow& uf)' or 'catch (MathErr& me)' if MathErr is base
```

Execution blocks



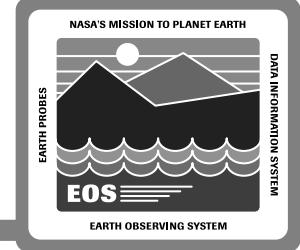


More about exceptions

- compiler manages ‘stack unwinding’ for automatic objects
- control is transferred to the nearest handler with ‘correct’ type
- if no exception is presently handled, “`terminate()`” is called
- an exception is considered “finished” when the corresponding catch handler exits
- the handlers are tried in order of appearance
- a base class handler will catch all exceptions thrown for the derived class (catch-handlers should be placed with care)
- control statements may be used to transfer control out of try or catch blocks but not into one.
- it is possible to list the exceptions thrown by a function

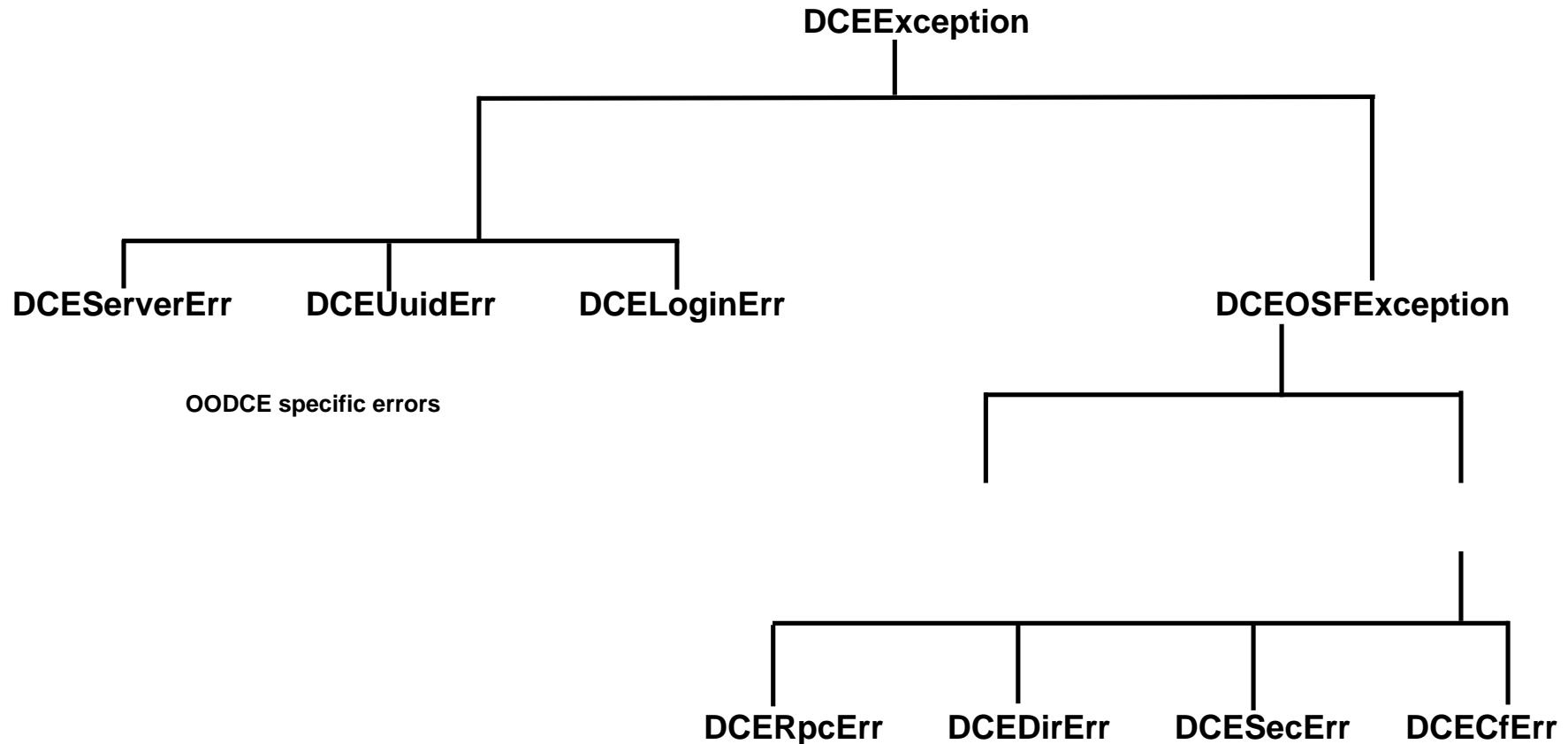
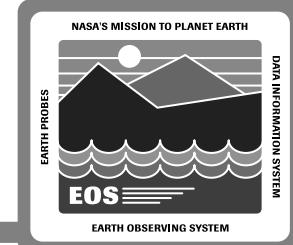
```
double Divide(...) throw(int, char *, Overflow);  
void foo() throw(); // no exception is thrown
```
- `unexpected()` is called if unspecified exception is thrown in a function

Example - HP OODCE

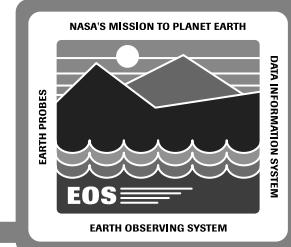


- **Uses exceptions for error notifications**
- **All calls to OODCE library should be in a try block**
- **Defines a exception class hierarchy**
- **Servers use exceptions to communicate errors to the clients**

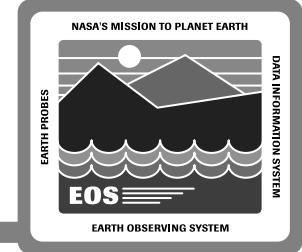
HP OODCE - Class hierarchy



OODCE Example (Server code)

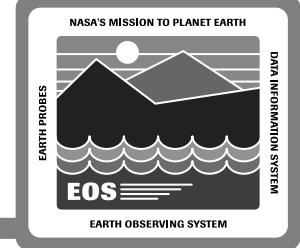


```
main ()  
{  
    try { //start of try block. notice that it has multiple statements and there is no error checking after  
        //each statement as we would have to do without exception handling  
        MyServer_1_0_Mgr* myMgr = new MyServer_1_0_Mgr; // instantiate server manager  
  
        DCEPthread* pth = new DCEPthread(theServer->ServerCleanup, NULL); // start signal handler  
  
        theServer->RegisterObject(myMgr); // register our manager with GSO  
  
        theServer->SetName("./:subsys/HP/sample-apps/MyManager"); // set name of the server in CDS  
  
        theServer->Listen(); // Listen for client requests  
    } // end of try block  
  
    catch (DCESecErr& err) { // catches all DCE security exceptions  
        cerr << "DCE Security error caught: " << (char*)err << endl; //class provides conversion to (char*)  
    } // end of DCESecErr catch handler  
  
    catch (DCEErr& err) { // catches all DCEErr exceptions thrown by the library  
        cerr << "DCE Error caught: " << (char*) err << endl; // DCEErr class provides conversion to (char*)  
    } // end of first catch handler  
  
    catch (...) // catch all  
    {  
        cerr << "unknown exception" << endl; // can't do much about it !!  
        throw; // rethrow esxception. will end up in terminate() since no higher level handler  
    }  
    // this is where we come after a successful try block or after catching the exception.  
    cout << "end of execution. manager exiting" << endl;  
}
```



Roguewave

- Robust, threadsafe foundation class library
- Provides classes for
 - time and date handling
 - internatiolization
 - managing collections, bit vectors, caching manager, virtual arrays
 - file management
 - stringsAND
 - error handling using exceptions
- Available on many unix platforms (also on PC)
- Implements exception handling using macros when not supported by the compiler (uses compiler supported scheme otherwise)
- Technical evaluation complete



Exception handling for ECS

Why

- modular, compact and less complex code
- consistent error handling
- scoping

What (is needed)

- Basic exception class hierarchy
- Developer awareness, usage guidelines (project instructions)
- Body to moderate issues and revise guidelines (CSWG or subgroup)

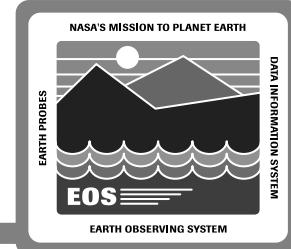
How

- All ECS libraries will use exceptions to return errors
- Applications will install several catch handlers for scoping
- Applications will be able to define specialized exceptions

Who

- ALL

Sample ECS Exception Class Hierarchy



Declaration

```
class ECSEException
{
    EcTLong ErrNo;
}

class ECSSystemExc:
    public ECSEException {
protected:
    char err_string[MAX-STRING]
}
```

Hierarchy

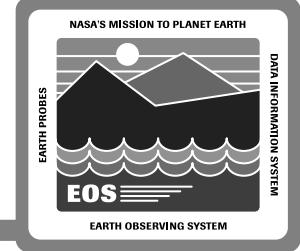
```
ECSEException
ECSSystemExc
ECSFileExc
ECSMemExc
ECSStreamExc
ECSProcessExc
ECSShareLibExc

ECSCommExc
ECSNetworkExc
ECSProtocolExc

ECSSecurityExc
ECSLoginExc
ECSAccessExc
ECSItruderExc

ECSDeviceExc
```

Sample ECS exceptions



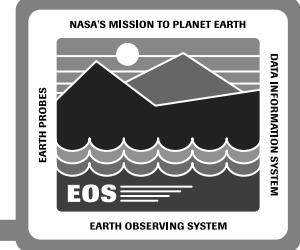
Clients

- **server down**
- **lookup failure**
- **security exception**
 - **authentication failure**
 - **access denied**

Servers

- **export failure**
- **registration failure**

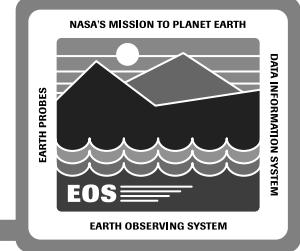
Summary



Exceptions

- provide an elegant method for error handling
- and error returns are mutually exclusive
- are supported by many compilers
- allow development of compact and less complex code
- are easy to deal with and use
- ARE FRIENDS !

File Transfer Interface



Objective

- Provide application interface to send/receive files.

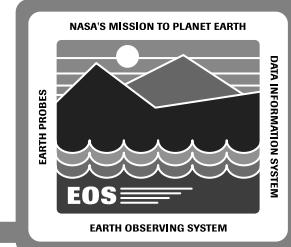
Features

- Uses interactive ftp client for transfers (one end must be local)
- Manages mapping of interactive messages to error codes
- Application interacts with a class
- Multiple file transfers allowed using the same instance

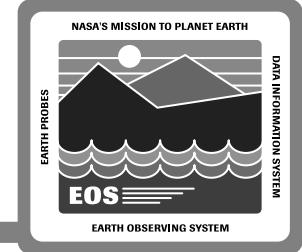
Issues

- Security
- Managing interactive client messages is difficult (write socket interface for the ftp protocol)

Object model



CsFtFTP
myRemoteHost: EcTChar*
myUserName: EcTChar*
myPassword: EcTChar*
CsFtFTP(EcTChar* RemoteHost , EcTChar* UserName = EcDNull, EcTChar* UserPassword = EcDNull)
Send(EcTChar* LocalFile, EcTChar* RemoteFile = EdDNull) : Ec TInt
SendAll(EcTChar* LocalDir=EcDNull, EcTChar* RemoteDir=EcDNull): Ec TInt
SendMatching(EcTChar* WildCard, EcTChar* RemoteDir=EdDNull): Ec TInt
Receive(EcTChar* RemoteFile, EcTChar* LocalFile = EcDNull) : Ec TInt
ReceiveAll(EcTChar* RemoteDir=EcDNull, EcTChar* LocalDir=EcDNull): Ec TInt
ReceiveMatching(EcTChar* WildCard, EcTChar* LocalDir=EdDNull): Ec TInt
SetRemotePath(EcTChar* path) : Ec TInt
GetRemotePath(): EcTChar*
SetLocalPath(EcTChar* path): Ec TInt
GetLocalPath(): EcTChar*
SetMode(CsEFtFtpMode Mode) : Ec TInt
GetHostName() : EcTChar*
SetUserName(EcTChar* UserName): Ec TInt
SetPassword(EcTChar* Password): Ec TInt
SetRemoteHost(EcTChar* HostName): Ec TInt



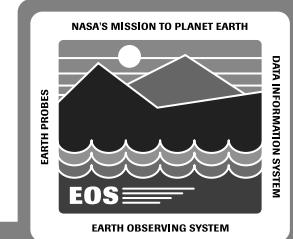
Example

```
// NOTE: no error handling shown here
#include <CsFtFTP.h> // include the class header
... // other stuff here

main()
{
    EcTInt err; // to receive errors from the class

    // instanciate the class. If "myName" and "myPassword" is not provided, anonymous ftp is assumed
    CsFtFTP* myFtp = new CsFtFTP("baltic", "myName", "myPassword");

    err = myFtp->Send("myLocalFile"); //copies myLocalFile to remote (login directory)
    ... // misc processing
    err = myFtp->Receive("/pub/data/aRemoteFile", "mydata/myLocalCopy"); // copy aRemoteFile
                                // with name myLocalCopy. full/partial names are allowed
    ... // misc processing
    err = SetRemotePath("datafiles/misc"); // sets remote path. all transfers will now be to/from that directory
    err = SendAll(); // sends all files in the current local directory to the current remote directory
    err = ReceiveAll("/pub/data/misc", "/home/data"); // copy all files from remote to local
    err = SendMatching("*.cxx"); // send all files matching *.cxx
}
```



Error Codes

ECsFtNoSuchHost - Remote host can't be found

ECsFtConnectFailed - Remote host down

ECsFtLoginFailed - Authentication failed (username/password not valid)

ECsFtNoSuchFile - File does not exist

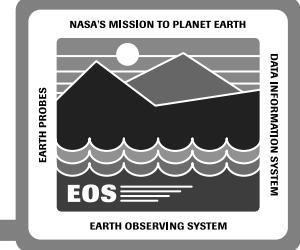
ECsFtInvalidArgument - Null/invalid argument provided to a function

ECsFtAccessDenied - Unable to complete operation

ECsFtNoMemory - Out of memory

ECsFtSystemError - System error in executing the ftp client

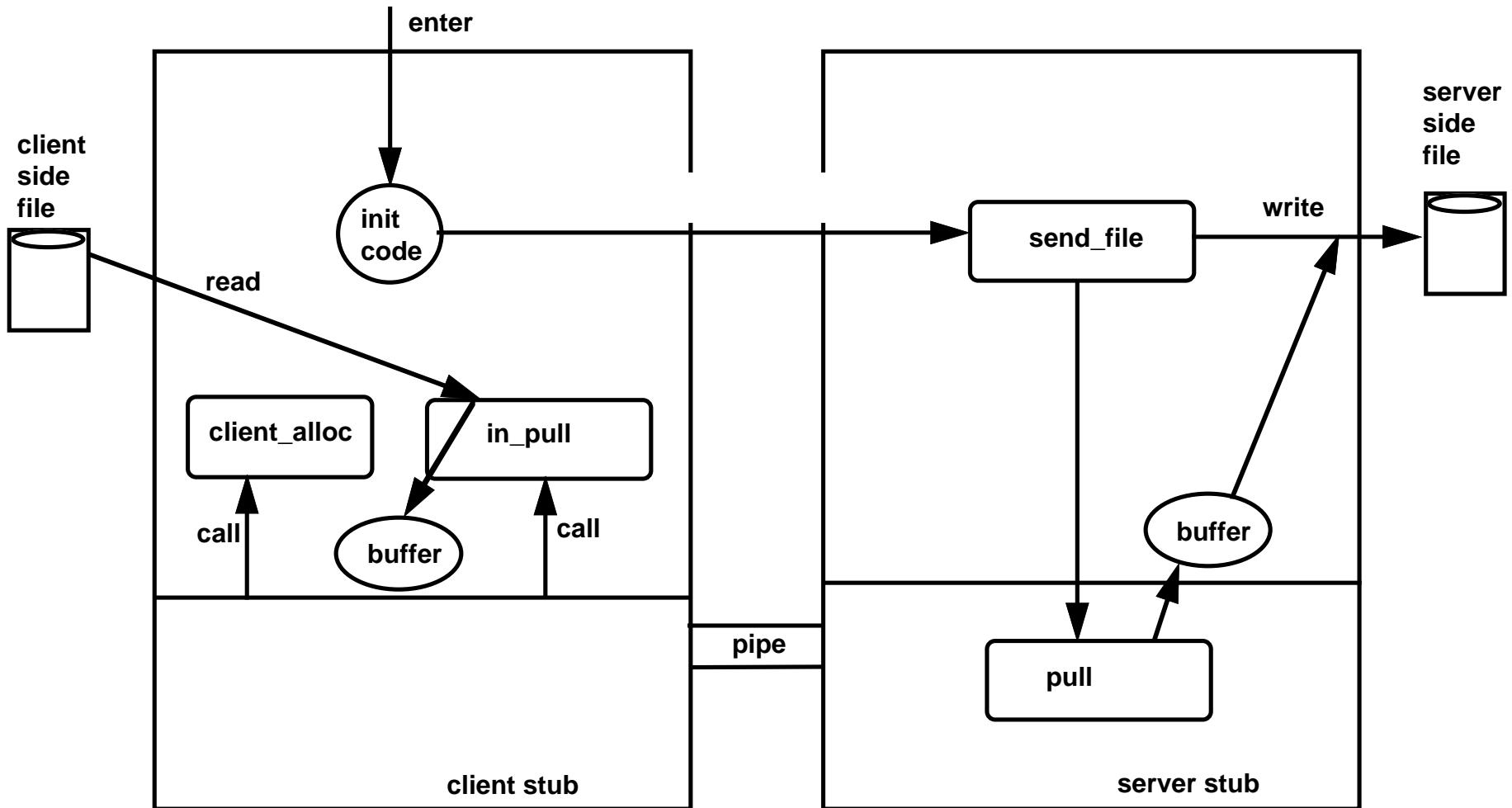
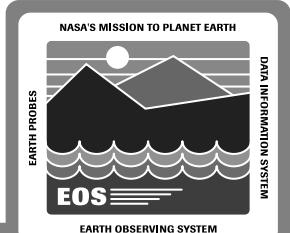
DCE RPC Pipes



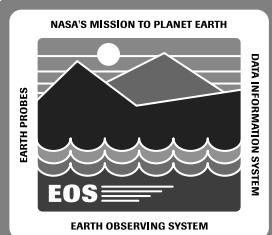
- Efficient way to transfer large (bulk) or incrementally produced data.
- Efficiency achieved by the RPC mechanism
- Defined in IDL (pipe type is available)
- As [in] parameter transfers data from client to server ([out] from server to client)
- Frame of reference is server. Pull is used for input pipe (C to S) and Push for output pipe (S to C)
- Client initiates transfer and server controls the loop
- DCE security is available

- Is another alternative (apart from ftp) for file transfers within ECS
- DCE is needed (external interfaces MAY NOT use it)
- File transfer servers (like ftpd) would be needed on all ECS hosts
- CSS prototyped DCE file transfer class

Input pipe (C to S) (Pull)



Mechanics - 1 (idl and header)



IDL

```
interface transfer_file
{
    typedef [handle] struct {
        char remote_host[200];
        char file_name[200];
    } cust_handle_spec; // a customized handle

    typedef pipe_char pipe_char;

    void send_file( // for in_pipe (pull by server)
        [in] cust_handle_spec cust_handle,
        [in] pipe_char data);

    void receive_file( // for out_pipe (push)
        [in] cust_handle_spec cust_handle;
        [out] pipe_char* data);
}
```

transfer_file.h (generated by IDL)

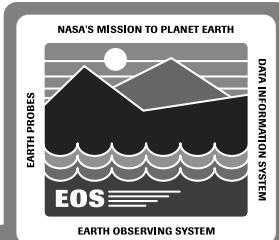
```
typedef struct pipe_char {
    void (*pull)(...);
    void (*push)(...);
    void (*alloc)(...);

    rpc_ss_pipe_state_t state; //DCE
}
```

client_pipe_state.h

```
typedef struct {
    int filehandle; // unix file handle
    char* filename; //name of the file
} client_pipe_state;
```

Mechanics - 2 (client code)



client main program

```
// Copy file /tmp/.cshrc from host kingkong
// to host fire as "/pub/testfile"

#include <transfer_file.h>
#include <client_pipe_state.h>

void client_alloc(); // in other module. called by stub
void in_pull(); // same as above

main()
{
    cust_handle_spec myHandle; // used by RPC
    client_pipe_state myPipeState; // used by client
    pipe_char myPipe; // pipe declaration
    // init handle for RPC
    myHandle.remote_host = "fire";
    myHandle.file_name = "/pub/testfile"; // remote name
    // init pipe state
    myPipeState.filehandle = -1;
    myPipeState.filename = "/tmp/.cshrc"; // local name
    // init pipe structure
    myPipe.state = (rpc_ss_pipe_state_t)&myPipeState;
    myPipe.alloc = client_alloc;
    myPipe.pull = in_pull;
    // initiate transfer
    send_file(myHandle, myPipe);
}
```

alloc function (called by stub to allocate buffer)

```
idl_char buffer[2000];

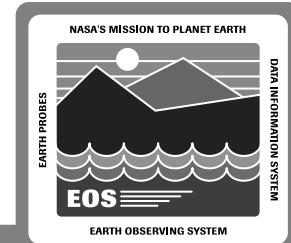
void client_alloc(
    client_pipe_state* state,
    idl_ulong_int bsize,
    idl_char** buf,
    idl_ulong_int* bcount)
{
    *buf = buffer; // give pointer to buffer
    *bcount = 2000; // size of buffer
    return; // to stub
}
```

in_pull function (called by stub to fill buffer)

```
void in_pull(
    client_pipe_state* state,
    idl_char* buf,
    idl_ulong_int maxsize,
    idl_ulong_int* count) // data size
{
    if (state->filehandle == -1) { open local file for read}
    // fill buffer with pipe data. set count
    *count = read(state->filehandle, buf, maxsize);

    if (*count == 0) // end of transfer
        close(state->filehandle);
    return; // to stub
}
```

Mechanics - 3 (server code)



Server Code

```
// called from the server main code.  
idl_char buffer[2000];  
  
void send_file( cust_handle_spec handle, // has name of the file to write  
                pipe_char          data)   // stub handles the pipe  
{  
    idl_ulong_int  count;  
    //open file for write  
    int file_handle = open(handle.filename, O_CREAT | O_TRUNC |  
O_WRONLY, 0777);  
    // loop until transfer is complete  
    while (TRUE)  
    {  
        (data.pull)(      // pull function in the server stub  
         data.state,    // state controlled in stub  
         buffer, 2000, // location and size of buffer to be filled  
         &count);     // stub returns number of chars filled in the buffer  
        // back from the stub.  
        if (count == 0) // transfer complete. end of client file  
            break; // end of loop  
  
        write(file_handle, buffer, count); // write to the file  
    }  
    close(file_handle);  
  
    return; // back to the main server program  
}
```

Execution blocks

